華中科技大學

课 程 实 验 报 告

课程名称：C++程序设计

实验名称：面向对象的整型栈编程

院　　系 ：计算机科学与技术

专业班级 ：　CS2006

学　　号 ：　U202015471

姓　　名 ：　杨释钧

指导教师 ：　纪俊文

2021 年 12 月 12 日

目录

1 需求分析

1.1 题目要求

整型栈是一种先进先出的存储结构，对其进行的操作通常包括：向栈顶压入一个整型元素、从栈顶弹出一个整型元素等。整型栈类 STACK 采用之前定义的两个 QUEUE 类模拟一个栈，其操作函数采用面向对象的 C++语言定义，请将完成上述操作的所有如下函数采用 C++语言编程， 然后写一个 main 函数对栈的所有操作函数进行测试，请不要自己添加定义任何新的函数成员和数据成员。

```
class STACK : public QUEUE {
    QUEUE q;
public:
    STACK(int m);                                   //初始化栈：最多存放 2m-2 个元素
    STACK(const STACK& s);                          //用栈 s 深拷贝初始化栈
    STACK(STACK&& s)noexcept;                       //用栈 s 移动拷贝初始化栈
    int    size()const noexcept;                    //返回栈的容量即 2m
    operator int() const noexcept;                  //返回栈的实际元素个数
    STACK& operator<<(int e);                       //将 e 入栈，并返回当前栈
    STACK& operator>>(int& e);                      //出栈到 e，并返回当前栈
    STACK& operator=(const STACK& s);//深拷贝赋值并返回被赋值栈
    STACK& operator=(STACK&& s)noexcept;//移动赋值并返回被赋值栈
    char * print(char *b)const noexcept;      //从栈底到栈顶打印栈元素
    ~STACK()noexcept;                               //销毁栈
};
```

编程时应采用 VS2019 开发，并将其编译模式设置为 X86 模式，其他需要注意的事项说明如下：

（1）在用 STACK(int m)对栈初始化时， 为其基类和成员 q 的 elems 分配 m 个整型元素内存，并初始化基类和成员 q 的 max 为 m，以及初始化对应的 head=tail=0。

（2）对于 STACK(const STACK& s)深拷贝构造函数，在用已经存在的对象 s 深拷贝构造新对象时，新对象不能共用 s 的基类和成员 q 为 elems 分配的内存，新对象要为其基类和成员 q 的 elems 分配和 s 为其基类和成员 q 的 elems 分配的同样大小的内存，并且将 s 相应的 elems 的内容深拷贝至新对象为对应 elems 分配的内存；新对象应设置其基类和成员 q 的 max、head、tail 和 s 的对应值相同。

（3）对于 STACK(STACK&& s)noexcept 移动拷贝构造函数，在用已经存在的对象 s 移动构造新对象时，新对象接受使用 s 为其基类和成员 q 的对应 elems 分配的内存，并且新对象的 max、head、tail 应和 s 的基类和成员 q 的对应值相同；s 的基类和成员 q 的 elems 设置为空指针以表示内存被移走，同时其对应的 max、head、tail 均应置为 0。

（4）对于 STACK& operator=(const STACK& s)深拷贝赋值函数，在用等号右边的对象 s 深拷贝赋值等

号左边的对象 s 时，等号左边对象的基类和成员 q 不能共用 s 的基类和成员 q 为 elems 分配的内存，若等号左边的对象为其基类和成员 q 的 elems 分配了内存，则应先释放掉以避免内存泄漏，然后为其 elems 分配和 s 为其基类和成员 q 的 elems 分配的同样大小的内存，并且将 s 对应两个 elems 的内容拷贝至等号左边对象对应两个 elems 的内存；等号左边对象中的 max、head、tail 应设置成和 s 中基类和成员 q 的对应值相同。

（5）对于 STACK& operator=(STACK&& s)noexcept 移动赋值，在用等号右边的对象 s 移动赋值给等号左边的对象时，等号左边的对象如果已经为其基类和成员 q 中的 elems 分配了内存，则应先释放以避免内存泄漏，然后接受使用 s 的基类和成员 q 为 elems 分配的内存，并且等号左边对象中的 max、head、tail 应和 s 中基类和成员 q 中的对应值相同；s 中基类和成员 q 的 elems 设置为空指针以表示内存被移走，同时其对应的 max、head、tail 均应设置为 0。

（6）栈空弹出元素或栈满压入元素均应抛出异常，并且保持其内部状态不变。

（7）打印栈时从栈底打印到栈顶，打印的元素之间以逗号分隔。

1.2 需求分析

本实验依托于实验二中设计的定长队列，要求使用两个队列来模拟栈这一个先进后出的数据结构，要求能够实现栈的所有基本操作。

2 系统设计

2.1 概要设计

利用两个队列模拟栈的面向对象的栈编程的总体思路是先用基类队列存放入栈元素，当基类队列满的时候，将入栈元素存放到实例数据成员队列中，出栈时首先从实例数据成员队列中取出元素，如果该队列为空，则从基类队列中取出元素，由于要实现栈的先进后出的特点，需利用循环将最后入栈的元素出栈。

本系统主要分为以下几个模块：

1. 队列的创建与赋值以及基本运算的实现；

2. 栈的创建与赋值以及基本运算的实现。

其中，栈作为队列的派生类，其功能实现直接调用队列的功能。类的头文件内容以及每一种方法的用途如下：

```
class QUEUE {
    int* const  elems;//elems申请内存用于存放队列的元素
    const  int  max;  //elems申请的最大元素个数为max
    int    head, tail;      //队列头head和尾tail，队空head=tail;初始head=tail=0
public:
    QUEUE(int m);       //初始化队列：最多申请m个元素
    QUEUE(const QUEUE& q);          //用q深拷贝初始化队列
    QUEUE(QUEUE&& q)noexcept;       //用q移动初始化队列
    virtual operator int() const noexcept;      //返回队列的实际元素个数
    virtual int size() const noexcept;          //返回队列申请的最大元素个数max
    virtual QUEUE& operator<<(int e);       //将e入队列尾部，并返回当前队列
    virtual QUEUE& operator>>(int& e);      //从队首出元素到e，并返回当前队列
    virtual QUEUE& operator=(const QUEUE& q);//深拷贝赋值并返回被赋值队列
    virtual QUEUE& operator=(QUEUE&& q)noexcept;//移动赋值并返回被赋值队列
    virtual char* print(char* s) const noexcept;//打印队列至s并返回s
    virtual ~QUEUE();                           //销毁当前队列
    };
class STACK : public QUEUE
{
    QUEUE q;
public:
    STACK(int m);                           //初始化栈：最多存放2m-2个元素
    STACK(const STACK& s);                  //用栈s深拷贝初始化栈
    STACK(STACK&& s)noexcept;               //用栈s移动拷贝初始化栈
    int   size()const noexcept;             //返回栈的容量即2m
    virtual operator int() const noexcept;          //返回栈的实际元素个数
    STACK& operator<<(int e);               //将e入栈，并返回当前栈
```

```
STACK& operator>>(int& e);              //出栈到e，并返回当前栈
STACK& operator=(const STACK& s);   //深拷贝赋值并返回被赋值栈
STACK& operator=(STACK&& s)noexcept;//移动赋值并返回被赋值栈
char* print(char* b)const noexcept; //从栈底到栈顶打印栈元素
~STACK()noexcept;                       //销毁栈
};
```

2.2 详细设计

设计每个模块的实现算法（处理流程）、所需的局部数据结构。具体介绍每个模块/子程序的功能、入口参数、出口参数、流程（图）等。

一、队列的创建与赋值以及基本运算实现

队列的数据成员包括指向用于存放队列的元素内存的整形指针 elem，申请的最大元素个数 max，指示队首与队尾的整形变量队列头 head 和尾 tail。函数成员及其说明如下：

(1)初始化队列：最多申请 m 个元素 QUEUE(int m)：先为 elem 申请一块可以存放 m 个整形元素的内存，然后将 max 设置为 m，此时便完成了队列的创建，最后将 tail 与 head 设为 0，表示初始状态队列为空，队列初始化完成。

(2)用 q 深拷贝初始化队列 QUEUE(const QUEUE& q)：深拷贝初始化要求新对象的 elems 需要分配和 q 为 elems 分配的同样大小的内存，并且将 q 的 elems 的内容深拷贝至新对象分配的内存；新对象的 max、head、tail 应设置成和 q 的对应值相同。因此先为 elems 分配一块大小为 q.max 大小的内存，然后将 max，head，tail 分别设为 q.max，q.head，q.tail，最后遍历 q 将 q 中所有元素赋给新对象的对应下标的元素。

(3)用 q 移动初始化队列 QUEUE(QUEUE&& q)noexcept：移动初始化要求新对象接受使用对象 q 为 elems 分配的内存，并且新对象的 max、head、tail 应设置成和对象 q 的对应值相同；然后对象 q 的 elems 设置为空指针以表示内存被移走，同时其 max、head、tail 均应设置为 0。因此直接将 elems 指向 q.elems，然后将 max，head，tail 分别设为 q.max，q.head，q.tail，最后将 q.elems 设为 nullptr，将 q.max，q.head，q.tail 设为 0。其中由于 elems 和 max 都是只读类型的变量，所以修改其值时需要先取地址，然后进行指针强制类型转换，最后再访问指针指向的内存即可。

(4)返回队列的实际元素个数 virtual operator int() const noexcept：由于队列时循环队列，故 tail 可能小于 head，因此队列的元素个数应为(tail+max-head)%max。考虑到 max 可能为 0，故只有再 max 不为 0 的时候返回(tail+max-head)%max，若为 0 则直接返回 0。

(5)返回队列申请的最大元素个数 virtual int size() const noexcept：直接返回 max 即可。

(6)将 e 入队列尾部，并返回当前队列 virtual QUEUE& operator<<(int e)：入队前先要判满，循环队列满的条件为若再入一个元素则 tail 与 head 相等，即(tail+1)%max==head。若队列满则抛出异常，若队列未满则将元素放入 elem 中下标为 tail 的内存，最后改变 tail，根据循环队列的规律，tail 应该改变为(tail+1)%max。

(7)从队首出元素到 e，并返回当前队列 virtual QUEUE& operator>>(int& e)：出队前先要判空，循环队

列空的条件为 tail==head。若队列空则抛出异常，若队列不为空则将下标为 head 的值赋给 e，然后改变 head 的值，根据循环队列的规律，head 的值应该改变为(head+1)%max。

(8)深拷贝赋值并返回被赋值队列 virtual QUEUE& operator=(const QUEUE& q)：首先若传入的队列若就是自己，即 elem==q.elem，则没必要进行后面的操作，直接返回*this 即可。若 this 也被初始化过，即 elem 不为空，由于之后要对其分配新的内存使其指向的内存大小与 q.elem 相同，为防止内存泄漏，先释放 elem 的内存将其设为空，后面的操作与深拷贝初始化一致。

(9)移动赋值并返回被赋值队列：QUEUE& QUEUE::operator=(QUEUE&& q) noexcept：首先若传入的队列若就是自己，即 elem==q.elem，则没必要进行后面的操作，直接返回*this 即可。若 this 也被初始化过，即 elem 不为空，由于之后要将其指向 q.elem，为防止内存泄漏，先释放 elem 的内存将其设为空，后面的操作与深拷贝初始化一致。

(10)打印队列至 s 并返回 s virtual char* print(char* s) const noexcept：基本思路是使用双指针 i，j，i 用来表示当前读入的队列的元素的标号，j 用来表示当前的字符串数组中的下标，采用 sprintf 函数进行打印，注意最后为了方便，其实在数组的最开始多打印了一个逗号，需要用 strcpy 函数将该逗号消除。

(11)销毁当前队列 virtual ~QUEUE()：释放 elem 内存后将数据成员全部设为 0 即可。

二、栈的创建与赋值以及基本运算的实现

栈 STACK 公有继承自 QUEUE，其数据成员除 QUEUE 的数据成员外，还有一个 QUEUE 类型的 q，基类的队列用于存放后入栈的元素，q 用于存放基类队列存放不下的先入栈的元素。函数成员及其说明如下：

(1)初始化栈最多存放 2m-2 个元素 STACK(int m)：由分析可知栈的全部操作都由两个队列的操作得到，故初始化栈就是初始化基类和成员 q，只需调用两者的形参为一个整形变量的初始化构造函数即可，传入 m，则每个队列可存放的元素数量为 m-1，栈可存放的元素数量为 2m-2。

(2)用栈 s 深拷贝初始化栈 STACK(const STACK& s)：由分析可知栈的全部操作都由两个队列的操作得到，故初始化栈就是初始化基类和成员 q，只需调用两者的形参为 QUEUE 类型的引用的初始化构造函数即可，对基类队列可以直接传入 s，最好将 s 强制转化为 QUEUE&，对 q 则需要传入 s.q。

(3)用栈 s 移动拷贝初始化栈 STACK(STACK&& s)noexcept：由分析可知栈的全部操作都由两个队列的操作得到，故初始化栈就是初始化基类和成员 q，只需调用两者的形参为 QUEUE 类型的右值引用的初始化构造函数即可，对基类队列可以直接传入 s，最好将 s 强制转化为 QUEUE&&，对 q 则需要传入 s.q，同样最好将其转化为 QUEUE&&。

(4)返回栈的容量 int size()const noexcept：两个队列的容量加起来就是栈的容量，故直接调用基类的 size()和 q 的 size()相加返回即可。

(5)返回栈的实际元素个数 operator int() const noexcept：两个队列中实际元素的个数和就是栈中元素的实际个数，故直接调用基类和 q 的 operator int()然后相加并返回即可。

(6)将 e 入栈，并返回当前栈 STACK& operator<<(int e)：根据概要设计中的思路，在基类队列未满的情况下，直接让元素进入基类队列中即可；若基类队列已满但 q 未满，则基类队列出一个元素，出来的元素即此时基类队列中最先入队的元素，让这个元素进入队列 q 中，然后再让新元素进入基类队列中，以此保证基类队列中的元素相较于 q 队列的都是后入栈的元素。若两队列都满，则抛出栈满的异常。

(7)出栈到 e，并返回当前栈 STACK& operator>>(int& e)：由于基类队列中的元素相较于 q 队列的都是后入栈的元素，所以当基类队列非空时，先出基类队列中的元素，而基类队列的队尾元素才是最后入栈的元素，我们真正要出的元素是队尾元素。设基类队列中元素的个数为 sizebase，则应该指向先将一个元素出队，在让其入队，以此不断将基类队列中队首元素移至队尾，执行 sizebase-1 次后，栈顶元素到了基类队列队首，将其出队至 e 即可。若基类队列为空而 q 队列不为空，则应该出 q 队列中的元素，出队的方式与上述基类队列出队方式一样。若基类队列与 q 都为空，则抛出栈空的异常。

(8)深拷贝赋值并返回被赋值栈 STACK& operator=(const STACK& s)：深拷贝赋值栈其实就是深拷贝赋值基类队列与 q 队列，直接调用两者的深拷贝赋值方法即可，传入的参数与深拷贝构造一致。

(9)移动赋值并返回被赋值栈 STACK& operator=(STACK&& s)noexcept：移动赋值栈其实就是移动赋值基类队列与 q 队列，直接调用两者的移动赋值方法即可，传入的参数与移动构造一致。

(10)从栈底到栈顶打印栈元素 char * print(char *b)const noexcept：两队列元素入队的顺序其实就是元素入栈的顺序，且在栈中 q 的队首元素即为栈底，基类的队尾元素即为栈顶，因此我们只需分别调用 q 与基类的 print 方法得到两个字符串，然后将两字符串连接，将保存有 q 的元素的字符串放在前面，保存有基类元素的字符串放后面。

(11)销毁栈~STACK()noexcept：销毁栈即销毁基类队列与 q 队列，分别调用二者的析构函数即可。

3 软件开发与测试

3.1 软件开发

硬件环境：

- 处理器：AMD Ryzen 5 4600H with Radeon Graphics 3.00 GHz
- 机带 RAM：16.0GB(15.4GB 可用)
- 系统类型：64 位操作系统，基于 x64 的处理器

开发环境：MSVC 2019 C++11，编译模式为 X86。 部分代码在 gcc， gdb 环境调试完成。

3.2 软件测试

测试结果如图 3.1 所示。

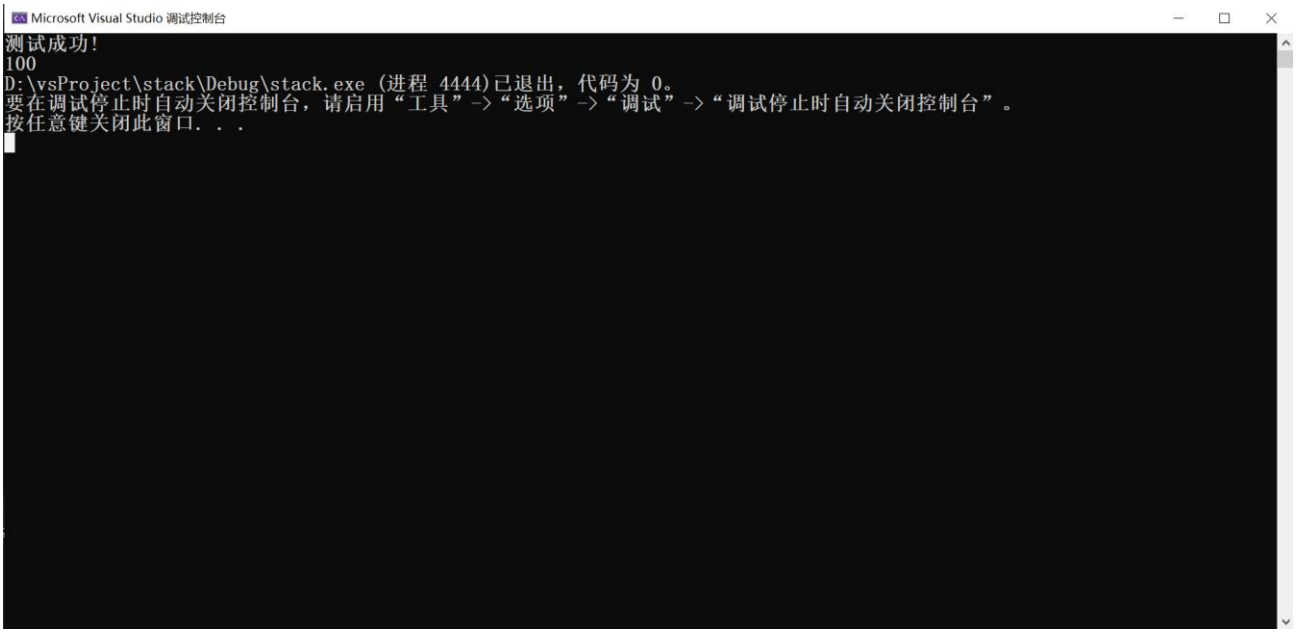

图 3. 1

测试结果较好。

4 特点与不足

4.1 技术特点

在继承了一个基类队列之后又包括了一个实例数据成员队列，使用两个队列模拟一个栈。

4.2 不足和改进的建议

队列为定长队列，泛用性差，可以改进为动态分配的内存的队列，即在队列满时调用 realloc 函数为队列重新分配一块更大的内存空间同时更新 max 的值，也可以使用链式队列，以提升空间利用率。

5 过程和体会

5.1 遇到的主要问题和解决方法

1. 本次实验中最先碰到主要的问题是如何用两个队列模拟一个栈，最开始我的想法是用一个队列来存放元素，另外一个队列作为辅助队列，在要出栈的时候将存放元素队列队尾元素前的元素全部出队并加入到辅助队列中，但是这样的空间利用率太低，并且不符合要求，于是考虑一些别的办法。而一个队列模拟一个栈很好想，入栈就直接入队，出栈就连续出队一个元素再将该元素入队直至最开始队尾元素变为队首元素，然后出队即可，因此用两个队列模拟一个栈时我的另外一个想法是将两个队列看作一个队列，但是题目要求压栈时默认先将元素进入基类队列中，若基类队列已满而队列 q 不满时，设法从基类队列挪出一个元素到 q，然后再往基类队列加入要压栈的元素。这一点想了很久没想明白，在经过和同学的讨论之后才意识到原来应该让一个队列保存后进栈的元素，另一个队列保存先进栈的元素，出栈时根据循环队列的特性使最晚入栈的元素出栈即可。

2. 另外其实就是在实验中对于 C++的一些特性不够熟悉，比如虚函数部分，在实验的时候出现了几次自递归的问题。

5.2 课程设计的体会

通过本次的实验让我对 C++中派生类问题有了更进一步的了解。刚开始进行编写时，对一些知识有点模糊，尤其是对于虚函数，总是容易弄混 QUEUE 和 STACK 中的一些同名函数，但经过老师和同学们的帮助下还是顺利的完成了。

6 源码和说明

6.1 文件清单及其功能说明

提交文件中，stackdef. h 文件中包含队列和栈的声明，在 queue. cpp 中包含队列相关函数的实现，在 stack. cpp 文件中包含栈相关函数的实现，在 main. cpp 文件中包含测试文件。

6.2 用户使用说明书

在 vs2019 中打开，并在本地运行。

6.3 源代码

stackdef.h:

```
#ifndef ABDC_H
#define ABCD_H

#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
class QUEUE {
    int* const elems;//elems申请内存用于存放队列的元素
    const int max; //elems申请的最大元素个数为max
    int head, tail;        //队列头head和尾tail，队空head=tail;初始head=tail=0
public:
    QUEUE(int m);        //初始化队列：最多申请m个元素
    QUEUE(const QUEUE& q);          //用q深拷贝初始化队列
    QUEUE(QUEUE&& q)noexcept;       //用q移动初始化队列
    virtual operator int() const noexcept;     //返回队列的实际元素个数
    virtual int size() const noexcept;         //返回队列申请的最大元素个数max
    virtual QUEUE& operator<<(int e);        //将e入队列尾部，并返回当前队列
    virtual QUEUE& operator>>(int& e);       //从队首出元素到e，并返回当前队列
    virtual QUEUE& operator=(const QUEUE& q);//深拷贝赋值并返回被赋值队列
    virtual QUEUE& operator=(QUEUE&& q)noexcept;//移动赋值并返回被赋值队列
    virtual char* print(char* s) const noexcept;//打印队列至s并返回s
    virtual ~QUEUE();                          //销毁当前队列
};
```

```
class STACK : public QUEUE
{
    QUEUE q;
public:
    STACK(int m);                          //初始化栈：最多存放2m-2个元素
    STACK(const STACK& s);                 //用栈s深拷贝初始化栈
    STACK(STACK&& s)noexcept;              //用栈s移动拷贝初始化栈
    int  size()const noexcept;             //返回栈的容量即2m
    virtual operator int() const noexcept;      //返回栈的实际元素个数
    STACK& operator<<(int e);              //将e入栈，并返回当前栈
    STACK& operator>>(int& e);             //出栈到e，并返回当前栈
    STACK& operator=(const STACK& s);      //深拷贝赋值并返回被赋值栈
    STACK& operator=(STACK&& s)noexcept;//移动赋值并返回被赋值栈
    char* print(char* b)const noexcept; //从栈底到栈顶打印栈元素
    ~STACK()noexcept;                      //销毁栈
};
#endif // !ABDC_H
```

queue.cpp:

```
#include"defstack.h"
QUEUE::QUEUE(int m) :elems(new int[m]), max(m)//初始化队列，最多申请m个元素
{
    tail = 0;
    head = 0;
}
QUEUE::QUEUE(const QUEUE& q) : elems(new int[q.max]), max(0)//用q深拷贝构造队列
{
    for (int i = 0; i < q.max; i++)
    {
        elems[i] = q.elems[i];
    }
    *(int*)&max = q.max;
    head = q.head;
    tail = q.tail;
}
QUEUE::QUEUE(QUEUE&& q)noexcept :elems(q.elems), max(q.max)//用q移动初始化队列
{
    head = q.head;
    tail = q.tail;
    q.tail = 0;
    q.head = 0;
    *(int**)&q.elems = NULL;
    *(int*)&q.max = 0;
}
QUEUE::operator int() const noexcept//获取队列中元素个数
```

```
{
    return tail-head<0?tail-head+max:tail-head;
}
int QUEUE::size() const noexcept//获取队列中能存放的最大数量
{
    return max;
}
QUEUE& QUEUE::operator<<(int e)//入队
{
    if ((tail + 1) % max == head)
    {
        throw("QUEUE is full!");
        return *this;
    }
    elems[tail] = e;
    tail = (tail + 1) % max;
    return *this;
}
QUEUE& QUEUE::operator>>(int& e)//出队
{
    if (head == tail)
    {
        throw("QUEUE is empty!");
        return *this;
    }
    e = elems[head];
    head = (head + 1) % max;
    return *this;
}
QUEUE& QUEUE::operator=(const QUEUE& q)
{
    if (this == &q)
    {
        return *this;
    }
    if (elems != NULL)//防止反复析构
    {
        delete[] elems;
    }
    *(int**)&elems = new int[q.max];
    for (int i = 0; i < q.max; i++)
    {
        elems[i] = q.elems[i];
    }
    *(int*)&max = q.max;
```

```
    head = q.head;
    tail = q.tail;
    return *this;
}
QUEUE& QUEUE::operator=(QUEUE&& q) noexcept
{
    if (this == &q)
    {
        return *this;
    }
    if (elems != NULL)//判断是否为空，非空再delete
        delete[] elems;
    *(int**)&elems = q.elems;
    *(int*)&max = q.max;
    head = q.head;
    tail = q.tail;
    *(int*)&q.max = 0;
    *(int**)&q.elems = NULL;
    q.head = 0;
    q.tail = 0;
    return *this;
}
char* QUEUE::print(char* s) const noexcept
{
    int i = head;
    int j = 0;
    while (i != tail)
    {
        j += sprintf(s + j, ",%d", elems[i]);
        i = (i + 1) % this->max;
    }
    strcpy(s, s + 1);//将第一个逗号去除
    return s;
}
QUEUE::~QUEUE()
{
    if (elems != NULL)
        delete[] elems;
    *(int**)&elems = NULL;
    *(int*)&max = 0;
    head = 0;
    tail = 0;
}
```

stack.cpp:

```
#include"defstack.h"
STACK::STACK(int m):QUEUE(m),q(m)
{}
STACK::STACK(const STACK &s):QUEUE(s),q(s.q)
{}
STACK::STACK(STACK&& s)noexcept :QUEUE((QUEUE&&)s),q((QUEUE&&)s.q)
{}
int STACK::size()const noexcept
{
    return QUEUE::size()+q.size();
}
STACK::operator int()const noexcept
{
    return int(q) + QUEUE::operator int();
}
STACK& STACK::operator<<(int e)
{
    if (int(q)+1 == q.size())
    {
        throw("STACK is full!");
        return *this;
    }
    if (QUEUE::operator int()+1 == QUEUE::size())
    {
        q<<(e);
    }
    else
    {
        QUEUE::operator<<(e);
    }
    return *this;
}
STACK& STACK::operator>>(int& e)
{
    if (STACK::operator int()==0)
    {
        throw("STACK is empty!");
        return *this;
    }
    if (int(q)==0)
    {
        int num = QUEUE::operator int();
        while (num > 1)
        {
            int front;
```

```
            QUEUE::operator>>(front);
            QUEUE::operator<<(front);
            num--;
        }
        QUEUE::operator>>(e);
    }
    else if(int(q)!=0)
    {
        int num = int(q);
        while (num > 1)
        {
            int front;
            q>>(front);
            q<<(front);
            num--;
        }
        q>>(e);
    }
    return *this;
}
STACK& STACK::operator=(const STACK& s)
{
    if (this == &s)
    {
        return *this;
    }
    QUEUE::operator=(s);
    q=(s.q);
    return *this;
}
STACK& STACK::operator=(STACK&& s)noexcept
{
    if (this == &s)
    {
        return s;
    }
    QUEUE::operator=((QUEUE&&)s);
    q=((QUEUE&&)s.q);
    return *this;
}
char* STACK::print(char* b)const noexcept
{
    QUEUE::print(b);
    strcat(b, ",");
    q.print(b + strlen(b));
```

```
    return b;
}
STACK::~STACK()
{}
```

main.cpp:

```
#include"defstack.h"
extern const char* TestSTACK(int& e);
int main()
{
    STACK s(10);
    int e;
    const char* ss = TestSTACK(e);
    std::cout << ss << std::endl;
    std::cout << e;
}
```